ECEC 412/621: Computer Architecture
Solution Key for Midterm Exam (Spring 2015–2016)

May 26, 2016

Student name:

The exam comprises of 4 questions and is out of 40 points.

1. **(10 points)** Consider the following snippet of MIPS code:

```
Loop:    ADD  $1, $2, $3
         SW   $zero, 0($1)
         ADDI $2, $2, #4
         BEQ  $2, $5, Loop
```

Assume that register \$3 contains the base address of the array and \$2 is initialized to zero. The register \$5 stores the loop bound. Unroll this loop once and schedule it for execution on a two-issue superscalar MIPS processor with forwarding enabled between the pipeline stages, including the delay slot, minimizing the number of execution cycles. Assume one delay slot per branch instruction. The branch instruction is resolved in the EX stage. You may also assume that any combination of instruction types can execute in the same clock cycle in the pipeline.

Solution: Let us unroll the above loop twice as follows:

```
Loop:    ADD  $1, $2, $3
         SW   $zero, 0($1)
         ADDI $2, $2, #4
         ADD  $4, $2, $3
         SW   $zero, 0($4)
         ADDI $2, $2, #4
         BEQ  $2, $5, Loop
```

Note that we have removed the name or anti-dependencies between the two `ADD` instructions—by using register \$4 for the second `ADD` and `SW` instead of \$1. Assume branches are resolved in the EX stage.

Clock cycle	**Instruction Slot 1**	**Instruction Slot 2**
1	`ADD $1, $2, $3`	`ADDI $2, $2, #4`
2	`ADD $4, $2, $3`	`ADDI $2, $2, #4`
3	`BEQ $2, $5, Loop`	`SW $zero, 0($1)`
4 (Delay slot)	`SW $zero, 0($4)`	`nop`

Note that since branches are resolved in the EX stage, result of the `ADDI` instruction issued in CC 2 can be forwarded to the `BEQ` instruction issued in CC 3. The delay slot is inserted in CC 4 and can be filled in by the `SW` instruction(s).

2. **(10 points)** Answer the following questions.

(a) Consider the following loop.

```
for (i = 1; i <= 100; i++){
    a[i] = b[i] + c[i];       // Statement S1
    b[i] = a[i] + d[i];       // Statement S2
    a[i + 1] = a[i] + e[i];   // Statement S3
}
```

List the dependencies between the statements in the loop and then rewrite the loop so that it is parallel. That is, loop iterations i and $i + 1$ should be able to execute in parallel.

Solution: We have a RAW dependency between statements S1 and S2 involving `a[i]`; a WAR dependency between statements S1 and S2 involving `b[i]`; and a RAW dependency between statements S1 and S3 involving `a[i]`.

Concerning parallelizing the above loop, let us examine two consecutive iterations of the loop:

Iteration 1:

```
a[1] = b[1] + c[1];
b[1] = a[1] + d[1]'
a[2] = a[1] + e[1]; // Statement S1
```

Iteration 2:

```
a[2] = b[2] + c[2]; // Statement S2
b[2] = a[2] + d[2]'
a[3] = a[2] + e[2];
```

Observe that statement S2 from the second iteration of the loop overwrites the value stored in `a[2]` by statement S1 during the first iteration. So, in general the value of `a[i + 1]` stored by S1 during loop iteration *i* will be immediately overwritten by S2 during loop iteration *i* + 1. So, we can clean up the loop as follows:

```
for (i = 1; i <= 100; i++){
    a[i] = b[i] + c[i];       // Statement S1
    b[i] = a[i] + d[i];       // Statement S2
}
a[101] = a[100] + e[100];
```

(b) Unroll the following loop four times and show the corresponding C code. Note that your code should work for arbitrary values of n (even when n is not a multiple of four).

```
for (i = 0; i < n; i++){
    a[i] = b[i] + c[i];
}
```

Solution:

```
multipleOfFour = floor(n/4);
leftOver = n - multipleOfFour*4;

for(i = 0; i < 4*multipleOfFour; i = i + 4){
    a[i] = b[i] + c[i];
    a[i + 1] = b[i + 1] + c[i + 1];
    a[i + 2] = b[i + 2] + c[i + 2];
    a[i + 3] = b[i + 3] + c[i + 3];
}

for(j = i; j < (i + leftOver); j++){
    a[j] = b[j] + c[j];
}
```

Better Solution:

```
nn = n - 4;

for(i = 0; i < nn; i = i + 4){
    a[i] = b[i] + c[i];
    a[i + 1] = b[i + 1] + c[i + 1];
    a[i + 2] = b[i + 2] + c[i + 2];
    a[i + 3] = b[i + 3] + c[i + 3];
}

for(j = i; j < (n - i); j++){
    a[j] = b[j] + c[j];
}
```

3. (**10 points**) Consider the following snippet of code:

```
d = 1;
do {
    for (i=0; i<5; i++) {                    /* branch b1 */
        r = generateRandomNumber();
        d = d + array[r];
        if (array[r] < 0)
            printf("Negative! \n");
        if (array[r] >= 0)                  /* branch b2 */
            printf("Not Negative! \n");
    }

    if (d < 0)
        printf("Not done yet \n");
    if (0 <= d < 20)
        printf("Getting closer \n");
    if (d >= 20)
        printf("We’re done! \n");           /* branch b3 */
    if (i == 5)                             /* branch b4 */
        printf("I am highly predictable. \n");
} while (d < 20)                            /* branch b5 */
```

Assume that the `generateRandomNumber()` function generates a random number greater than or equal to zero. Local and global history branch predictors are designed to take advantage of certain properties of branches. What kind of branch predictor, local or global history, are each of the branches—b1 through b5—in the above code fragment best suited for? In other words, which branch predictor is designed to take advantage of the properties exhibited by the branches? Explain your answers clearly and concisely. Assume that the `if` and `while` statements are implemented as branches.

1. Branch b1: Local predictor will capture the repetitive branch pattern of 111110, where 1 is taken and 0 is not taken.

2. Branch b2: A correlating or global predictor will capture the fact that the behavior of branch b2 is correlated with that of the branch immediately above it.

3. Branch b3: A correlating or global predictor will capture the fact that the behavior of branch b3 is correlated with those of the two previous branches.

4. Branch b4: Global history that looks at the outcome of branch b1.

5. Branch b5: Global history.

4. Consider the following snippet of code

```
ADDD  F2, F0, F2
MULTD F3, F0, F2
SUBD  F1, F4, F0
MULTD F3, F1, F2
ADDD  F1, F2, F5
```

that is to be dynamic scheduled using a scoreboard. The processor has six functional units with the following execution stage latencies:

Functional unit	Latency in cycles
ALU integer	1
FP add/subtract	5
FP add/subtract	5
FP multiply	10
FP multiply	10
FP divide	25

(a) **(10 points)** At the start of the program, no register values are being computed by any functional units (that is, all registers are ready). Suppose the first instruction `ADDD` issues on cycle 1 and reads operands on cycle 2. Show the status of each of the above instructions in the pipeline at the end of cycle 8.

Instruction status				
Instruction	Issue	Read operands	Execution	Write result
`ADDD F2,F0,F2`	CC 1	CC 2	CC 3-7	CC 8
`MULTD F3,F0,F2`	CC 2	CC 8		
`SUBD F1,F4,F0`	CC 3	CC 4	In progress	
`MULTD F3,F1,F2`				
`ADDD F1,F2,F5`				

Note the cycle number(s) during which the instruction passes through each stage. I have filled in two entries in the table as examples. If an instruction has stalled in the pipeline or has not yet been issued, point out the reason why, including the hazard involved.

1. `MULTD` can be issued in CC 2. However, it cannot read operands yet until CC 8 when `ADDD` writes results.
2. The `SUBD` is in progress and will finish execution in CC 9.
3. The second `MULTD` cannot be issued due to a WAW hazard involving the previously issued `MULTD`.
4. The final `ADDD` cannot be issued since the `MULTD` before it has not been issued.

(b) **(10 points)** Show the status of the scoreboard below at the end of cycle 8.

Scoreboard status									
Name	Busy	Opcode	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Add1	No								
Add2	Yes	`SUBD`	F1	F4	F0			No	No
Mult1	Yes	`MULTD`	F3	F0	F2			No	No
Mult2	No								
Divide	No								

Register result status									
	F0	F1	F2	F3	F4	F5	F6	. . .	F30
Functional unit		Add2		Mult1					

1. Note that the scoreboard entry for Add1 has the Rj and Rk flags set to No since the `ADDD` instruction has already read both its operands.
2. Note that the scoreboard entry for Add2 has the Rj and Rk flags set to No since the `SUBD` instruction has already read both its operands.